Electronic Communications of the EASST
Volume X (2010)

Proceedings of the
4th International Workshop on
Multi-Paradigm Modeling
(MPM 2010)

A generic in-place transformation-based approach to structured model co-evolution

Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle

13 pages

Guest Editors: Vasco Amaral, Hans Vangheluwe, Cécile Hardebolle, Lazlo Lengyel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A generic in-place transformation-based approach to structured model co-evolution

Bart Meyers[1], **Manuel Wimmer**[2], **Antonio Cicchetti**[3], **and Jonathan Sprinkle**[4]

[1] University of Antwerp, Belgium
bart.meyers@ua.ac.be
[2] Vienna University of Technology, Austria
wimmer@big.tuwien.ac.at
[3] Mälardalen University, MRTC, Västerås, Sweden
antonio.cicchetti@mdh.se
[4] University of Arizona, United States
sprinkle@ece.arizona.edu

Abstract: In MDE not only models but also metamodels are subject to evolution. More specifically, they need to be adapted to correct errors, support new and/or update language features. The direct consequence of such evolutionary steps comprises the problem of managing the co-evolution of existing model instances, which may no longer conform to the new metamodel version. This model migration is intrinsically complex and results in a time-consuming and error-prone process if no adequate support is provided. For tackling this problem, we introduce a new technique to guide the user in solving migration issues in a step-wise manner. The aims are manifold, notably the simplification of the migration specification, the reduction of the effort for the evolver, the control of user intervention, and the optimization of the migration execution itself by allowing in-place adaptation of the existing instances.

Keywords: Metamodel evolution, model co-evolution, in-place transformations

1 Introduction

In Model-Driven Engineering (MDE) not only models but also metamodels are subject to evolution. Especially, when domain-specific modeling languages are employed, the necessity of language adaptations arise to reflect changes in the modeling domain as well as in technologies without losing existing models.

Figure 1 illustrates the context of this paper at a glance. Full arrows are transformations, dashed arrows are conformance relations. After evolution Δ of a metamodel MM_L, the goal is to migrate models m, conform to MM_L, to m', conform to $MM_{L'}$, by creating a suitable migration M. Thus, (i) dedicated co-evolution languages, like COPE [HBJ09] and (ii) the usage of model-to-model (M2M) transformation languages [CDEP08] have been proposed to migrate models. However, this requires to learn a new language in the first case and to employ a heavyweight technique in the second case. Furthermore, currently there exists no approach which allows the step-wise migration of models in combination with systematically modeling the evolution, i.e., ensuring that the migrated models actually conform to the new metamodel version.

In this paper, we introduce a new approach to guide the user in solving co-evolution issues in a structured, step-wise manner. First, instead of describing the migration of models as a transformation between two metamodels, we employ existing in-place transformation languages for this task. As opposed to M2M transformations, in-place transformations are transformations that change the input model instead of creating an output model from scratch [KMS+09]. For being able to employ in-place transformations, the prerequisite is to represent both language versions within one metamodel which is automatically computed by merging MM_L and $MM_{L'}$. Second, we distinguish between syntactic and semantic migration. For *syntactic migration*, the goal is to make model instance syntactically conform to the new version of the metamodel. Syntactic migration is fully automatable and might therefore be the preferred choice. Often however, some metamodel changes are introduced with a specific purpose, which calls for a particular migration scenario for the instance models. In such cases, we speak of *semantic migration*. Semantic migration requires manual adaptation from the evolver. Third, for dividing-and-conquering the co-evolution process, we formalize metamodel evolution as a difference model consisting of a sequence of simple difference operations. For each difference operation or meaningful group of difference operations (defined by the evolver), a migration is either automatically generated or adapted by the user. By this, the model m evolves to m' over intermediate models m_i, conform to intermediate metamodels MM_i. Fourth, by computing for each step also a specific merged metamodel to allow the systematic modeling of in-place transformations and by using a so-called checkout transformation, we can ensure that each m_i conforms to MM_i, thus after all steps each migrated model m' is always conform to the $MM_{L'}$.

$$\begin{tikzcd}
\mathrm{MM_L} \arrow[r, "\Delta"] & \mathrm{MM_{L'}} \\
\mathrm{m} \arrow[u, dashed] \arrow[r, "\mathrm{M}"'] & \mathrm{m'} \arrow[u, dashed]
\end{tikzcd}$$

Figure 1: Models m have to be migrated when MM_L evolves.

The benefits of this technique are manifold, notably the simplification of the migration specification by reusing the well-known graph transformation formalism of in-place transformations, the ability to express every possible evolution and migration by allowing graph transformation techniques, the reduction of the effort for the user by reusing generically applicable migration rules, the control of user intervention by automated preventive and corrective mechanisms for conformance checking in each migration step, and the optimization of the migration execution itself by allowing in-place adaptation of the existing instances.

The remainder of this paper is structured as follows. Section 2 introduces a co-evolution scenario in the domain of rail road management which is used as running example throughout the paper. Section 3 presents our approach by first showing how the deltas between the metamodels are classified and represented, and second how the migration rules are derived from these deltas. Related work is discussed in Section 4, and finally, the paper is concluded with an outlook on future work in Section 5.

2 Example

In order to illustrate our approach, we first introduce an evolution scenario on the RailRoad domain-specific language. A RailRoad model is shown in Figure 2. The model can be used to analyze the behavior of trains riding on the modeled railtrack.

A RailRoad model consists of track elements, on which trains can ride. These elements can be either rails, which point to one other element on the track, or junctions, which point to two different elements on the track. In this example, two trains are riding on a track with one junction, and one train is not located on the track. The syntax of the RailRoad language is captured in its metamodel, shown in Figure 3a. A *Train* can be located *On* a *TrainPlace*, which can be a *Rail* or a *Split*. *Rail*s have one *Link* to another *TrainPlace*, *Split*s have two. These links are obligatory, so a RailRoad circuit is always closed.

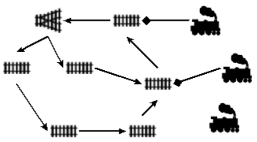
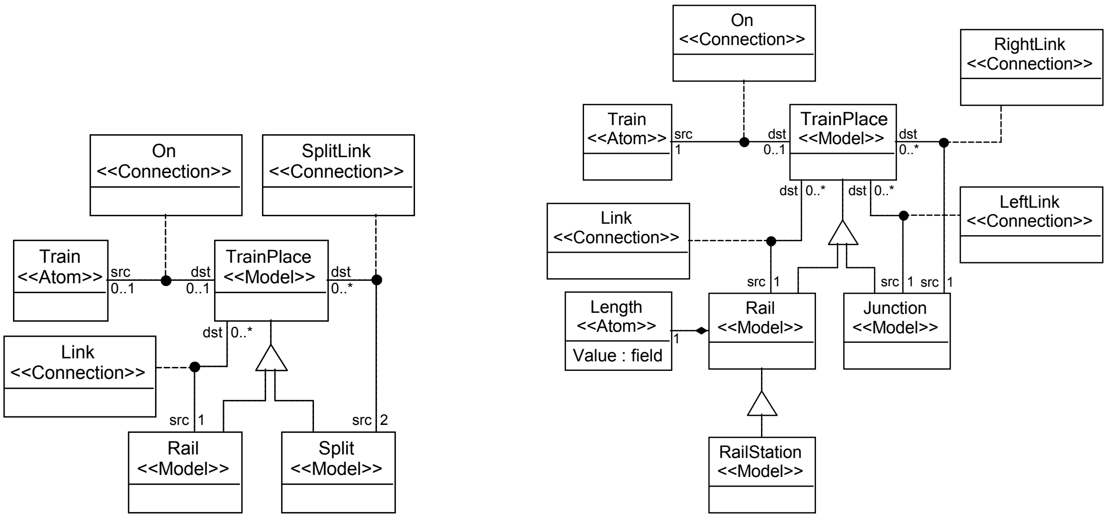


Figure 2: An example RailRoad model.



(a) Existing metamodel.

(b) Evolved metamodel.

Figure 3: (a) The existing metamodel, and (b) the evolved metamodel.

Suppose that over time, some changes have been applied to the metamodel. Five requirements are implemented. For each requirement it is stated how existing models should be migrated:

- *Split* has been renamed to the more intuitive “Junction.” In the instance models, each existing *Split* has to become a *Junction*;
- *Trains must* be on a *TrainPlace* now. In the instance models, *Train*s that are not located on a *TrainPlace* have to be removed;
- a notion of direction is added: instead of two outgoing *SplitLink*s, a *Junction* now has a *LeftLink* and *RightLink* direction. In the instance models, the two outgoing links to *TrainPlace*s must be replaced with a *LeftLink* and *RightLink* link. The choice of left and right is made randomly;
- a notion of track length has been added to a *Rail*. In the instance models, *Rail*s have a length of 1, the default length;
- a *RailStation* is introduced as a new kind of *Rail*. In the instance models, *Rail*s with more than one incoming *Link* or *SplitLink* are interesting places to build a *RailStation*.

The resulting metamodel is shown in Figure 3b[1]. In the remainder of this paper, this evolution scenario will be used to illustrate our structured migration approach.

3 Approach

Whenever a change Δ is operated on a metamodel, a corresponding migration M should be operated on the existing instances. The creation of migration transformation is closely related to the changes on the metamodel however. Therefore, this section starts off with an elaboration on the difference model, which is a structured representation of the changes. Next, the creation of the migration transformation is presented.

3.1 Difference model

A number of works proposed the classification of metamodel changes with respect to the effects observable for migration [GKP07, CDEP08]; in particular, the changes could require either no migrations of the instances (non-breaking operations), or simple migration adaptations (breaking and resolvable operations), or complex migrations which possibly require user input (breaking and unresolvable operations). As migration is directly linked to the metamodel changes, the migration transformation can be created from a difference model representing the evolution of the metamodel. In turn, the difference model is a sequence of *difference operations*, each of which mapping onto a corresponding *migration operation*, as summarized in Table 1.

Difference operation	Migration operation
Non-breaking operations	
Generalize metaproperty	None
Add non-obligatory metaclass	None
Add non-obligatory metaproperty	None
Extract superclass	None
Breaking and resolvable operations	
Eliminate metaclass	Eliminate instances
Eliminate metaproperty	Eliminate instances
Push metaproperty	Eliminate properties from superclass instances
Flatten hierarchy	Eliminate superclass instances
Rename metaclass	Change instances
Rename metaproperty	Change instances
Breaking and unresolvable operations	
Add obligatory metaclass	Add default instances
Add obligatory metaproperty	Add default instances
Pull metaproperty	Add default properties for superclass instances
Restrict metaproperty	Remove instance if non-compliant

Table 1: Difference operations based on [CDEP08], with their migration operations.

The evolutions listed in Table 1 represent manipulations that typically occur on a given metamodel, like the addition of a new metaclass (Add non-obligatory metaclass), the deletion of

---

[1] The field attribute for length is a type-safe integer, though this is not shown in the diagram due to the concrete syntax choices of the GME (Generic Modeling Environment) metamodeling paradigm

an existing metaattribute (Eliminate metaproperty), the rename of an element (Rename metaclass/metaproperty), and so forth. Beside such primitive operations, the table also lists complex evolutions like Flatten hierarchy (eliminating a superclass and adding all its properties to the subclasses) or Generalize metaproperty (relaxing the cardinality of a property); in those cases, the evolution could also be seen as the composition of simple changes, but it reaches its full meaning when considered as a single adaptation step. For instance, Flatten hierarchy flattens the metaclasses involved in a generalization relationship by moving all the existing metaattributes in a selected surviving metaclass and by eliminating all the remaining metaclasses and generalization relationships. Analogously, Pull metaproperty moves a metaproperty from a set of subclasses to their corresponding superclass. It is important to note that all possible changes to a metamodel can be represented by the difference operations of Table 1.

If the metamodel contains static semantics, in the form of e.g., OCL constraints [Obj10], similar operations can be contrived. OCL constraints for instance are conform to a fine-grained metamodel [RG99]. Therefore, one can reason (thus create operations) about adding, updating and deleting elements of instances, i.e., constraints, up to the detail of variables, expressions and values.

The classification proposed above highlights the criticality of the metamodel evolution detection and representation in order to achieve a profitable migration of the existing instances. Currently, (meta-)model comparison is an active field of research; it is an intrinsically complex task since it has to deal with graph isomorphisms, i.e., with the problem of finding correspondences between two given graphs. In this paper we assume that the metamodel evolution, i.e., Δ in Figure 1, is given, as reflecting the developer intentions, in terms of the operations classified in Table 1: it could be obtained as directly traced from a tool, or encoded by hand. For our approach, both techniques are applicable.

When the difference operations of Table 1 are used for the change Δ of the RailRoad example, this results in the difference model in Table 2, which is a sequence of method calls. The difference operations are instantiated as method calls, based on the operations of [HBJ09], which are predefined migration operations that take some parameters as input. When such a method is executed on the metamodel, the change is applied. Note that operations δ_3, δ_4 and δ_5 represent the replacement of *SplitLink* to *SplitLeft* and *SplitRight*. Other representations, such as proper difference languages [CDP07], can be used as well in our approach.

nr.	Operation
δ_1	RenameMetaElement(Split, “Junction”)
δ_2	RestrictMetaProperty(Train.On, 1, 1)
δ_3	EliminateMetaProperty(Junction.SplitLink)
δ_4	AddNonObligatoryMetaProperty(Junction, TrainPlace, “LeftLink”, 1, 1, 0, -1, False)
δ_5	AddNonObligatoryMetaProperty(Junction, TrainPlace, “RightLink”, 1, 1, 0, -1, False)
δ_6	AddObligatoryMetaProperty(Rail, “length”, Integer, 1, 1, 1)
δ_7	AddNonObligatoryMetaClass(”RailStation”, [Rail], False)

Table 2: The difference model Δ of the RailRoad evolution.

3.2 Migration of Instance Models

In this section it is explained how the instance models are migrated. With our approach, we aim at a high degree of automation, a high degree of control, and high execution performance. Automation will reduce the effort for the modeller, thus increase productivity. Control will increase correctness of the migration process as well as facilitate the migration process for the evolver. Performance will affect scalability, or the number of instance models that can be migrated in a given time. The migration process consists of three phases: automated synthesis, manual adaptation and execution.

3.2.1 Synthesis

In the first phase, we synthesize migration transformations from difference operations. This is done automatically, by generating an instance of the default migration transformation for each difference operation corresponding to Table 1. Note that the default migration transformation can be *None*, i.e., the identity transformation. Moreover, despite a metamodel manipulation could entail multiple migration policies, in general the default one is fixed once for all due to coherence purposes. The left part of Figure 4 shows the evolution Δ of the RailRoad example, split up into the seven δ_i, as shown in Table 2. In this step-wise approach, the metamodel MM_L evolves to $MM_{L'}$, over intermediate metamodels MM_i. For each δ_i, a μ_i is synthesized by applying the transformation G. Technically, G is a higher order transformation, because it takes transformation models instead of instance models as input or output [TJF+09]. The instance model m is migrated accordingly to m'. In this case, $MM_7 = MM_{L'}$ and $m_7 = m'$. The right side shows one generic migration step, where a metamodel MM_{i-1} evolves to MM_i by applying one difference operation. m_{i-1} is migrated accordingly.

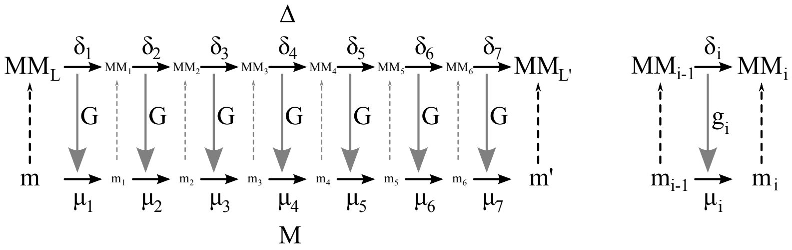


Figure 4: Synthesis of migration transformation μ_i (left) and one generic migration step (right).

Figure 5 shows an example of the synthesis of the migration operation μ_2. The migration operation is created from difference operation δ_2 (shown on top) and the template (shown on the left). The template for the migration operation for the “restrict metaproperty” difference operation is shown. The template specifies the default migration behavior: instances that do not conform to the evolved, more restricted, metamodel are removed. Removal is denoted by the “X” symbol on the element. The generic template of the migration transformation rule on the left side is completed with the information provided by the parameters of the difference operation on the top side. The resulting migration transformation rule on the right side deletes *Train*s that do not have exactly one *On* link. Note that the resulting rule is an in-place transformation rule,

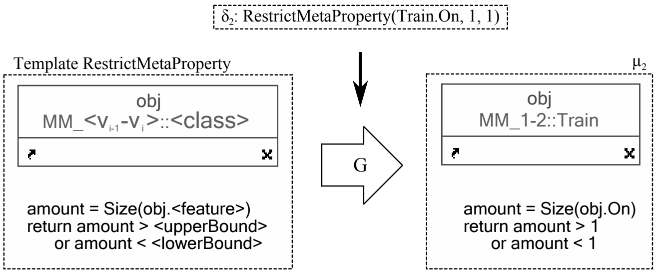


Figure 5: Creating a migration transformation μ_2 from difference operation δ_2 and a transformation template using merged metamodel $MM_{1,2}$.

and no model-to-model transformation. The in-place transformation captures only the essence of the migration problem, lowering the degree of accidental complexity.

In order to allow in-place transformation, the metamodels of the source- and target models must be the same. In our case, the metamodels MM_{i-1} and MM_i are very similar but not the same. Therefore, we *merge* both metamodels into one metamodel $MM_{i-1,i}$, to which models m_{i-1} as well as models m_i conform. $MM_{i-1,i}$ can be automatically generated from MM_{i-1} and δ_i, so that $MM_{i-1,i} = merge(MM_{i-1}, \delta_i)$: initially, $MM_{i-1,i} = MM_{i-1}$. If δ_i is additive, the change is applied to $MM_{i-1,i}$. If δ_i is subtractive, the to be deleted element is kept in $MM_{i-1,i}$. If δ_i is updative, the updated version is added to $MM_{i-1,i}$ without removing the old version. No matter what kind of change, the metamodel is “relaxed” so that all possible m_{i-1} and m_i are conform to $MM_{i-1,i}$. This is in particular important for obligatory changes, which are made non-obligatory in the merged metamodel by relaxing the cardinality of the involved associations. Figure 6 shows the merged metamodel $MM_{5,6}$ that is used for the migration transformation μ_6 that implements the introduction of the *Length* attribute. Notice that all changes δ_1 to δ_5 are already carried through, as migration step 6 is reached. δ_7 is disregarded for now, as this step is not reached yet. δ_6 is an additive change, so the new element, i.e., the *Length* feature, is added to the merged metamodel. Additionally, the cardinality is relaxed so that the *Length* feature is not obligatory.

Once the default migration operation is synthesized for each δ_i, the instance models m can be migrated by executing the sequence of in-place transformations $M = \mu_i \circ \mu_{i-1} \circ \ldots \circ \mu_2 \circ \mu_1$ of Figure 4. By construction, the resulting $m' = M(m)$ will syntactically conform to $MM_{L'}$.

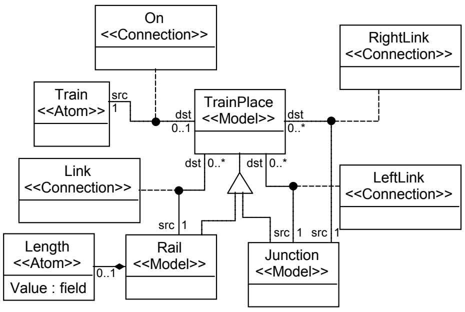


Figure 6: The merged metamodel $MM_{5,6}$ used for μ_6.

3.2.2 Manual adaptation

Technically, the first phase fulfills the requirement for co-evolution, namely restoring consistency in the conformance relation. Syntactic migration is thereby accomplished. In the RailRoad evolution, however, there are also cases of semantic migration. Examples are the introduction of the notion of direction and the introduc-

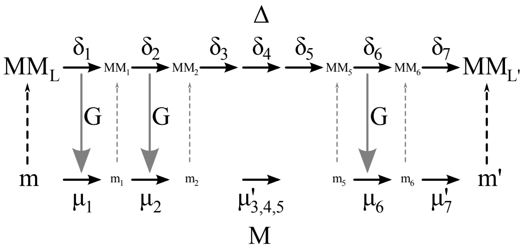


Figure 7: The step-wise migration after the manual adaptation phase.

tion of the *RailStation*. Semantic migration is done during the manual adaptation phase.

In this phase, each δ_i and corresponding default μ_i are one by one presented by the evolver. For each μ_i, the evolver can choose from four possible actions:

- *keep* the default μ_i. If the evolver is satisfied with the default μ_i, nothing has to be done for this step. This action is typically applied for non-breaking or breaking and resolvable changes;
- *edit* the default μ_i. The evolver might be satisfied with the structure of the default μ_i, but might wish to alter μ_i slightly to $\mu_i^{'}$. This action is typically applied for breaking and resolvable changes or breaking and unresolvable changes;
- *group* the current μ_i with following μ_{i+1}. In some cases, a number of difference operations can be grouped as one conceptual change, requiring one $\mu_S^{'}$ (with S a sequence of consecutive indices) for two or more difference operations;
- *create* a tailored migration for the corresponding difference operation. If the evolver has a migration transformation in mind that is completely different than the default one, he/she can create his/her own. The action is typically applied for non-breaking (if the evolver actually wants to migrate instead of doing nothing) or breaking and unresolvable changes. Note that by first grouping and next creating, the original μ_i and μ_{i+1} are replaced by one $\mu_{i,i+1}^{'}$ that covers both the migration of δ_i and δ_{i+1}. Also note that so-called model specific migration can be introduced here, requiring user input at migration time [HBJ09].

Figure 7 shows the result of the RailRoad migration after the manual adaptation phase. μ_1, μ_2 and μ_6 are kept, $\mu_7^{'}$ is created manually and δ_3, δ_4 and δ_5 have been grouped (introducing the notion of direction) and $\mu_{3,4,5}^{'}$ is created manually.

Figure 8 shows the custom migration transformation $\mu_{3,4,5}^{'}$. Two *SplitLink*s are replaced by a *LeftLink* and a *RightLink*, which covers the migration of the three changes δ_3, δ_4 and δ_5.

A new problem arises when allowing the evolver to manually create migration operations. After this phase, it cannot be guaranteed anymore that m' is conform to MM', as the evolver is allowed to implement anything he/she wants in the customized migration transformations. In our framework, we offer a solution to uphold this guarantee by providing maximal control over the creation of the migration operation, while still offering full expressiveness. This control is provided by two mechanisms, a preventive mechanism and a corrective mechanism:

- *restricted metamodel*: as a preventive mechanism, it is only allowed to use language constructs of the corresponding difference operation(s) when editing or creating a $\mu_S^{'}$ (with S

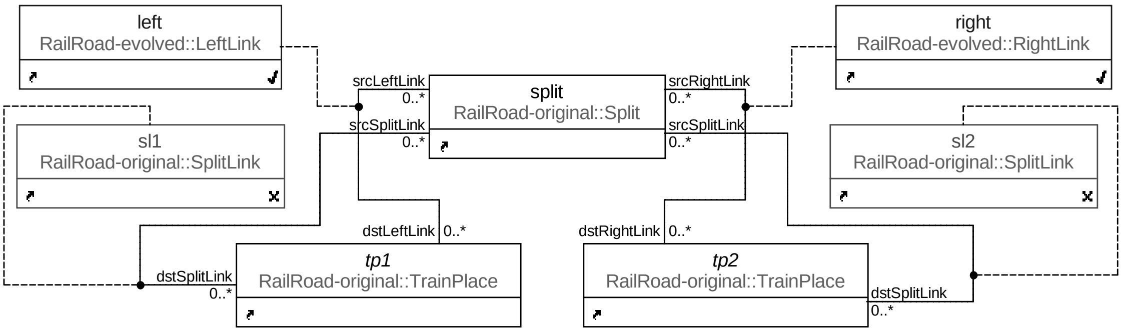


Figure 8: The customized migration transformation $\mu'_{3,4,5}$ introducing the notion of direction.

a sequence of one or more consecutive indices, though in many cases this is just one index x, as suggested in Figure 7). This means that we consider only a part of the total evolution for this migration, particularly $MM_{min(S)-1}$ to $MM_{max(S)}$ (in the case of $|S| = 1$ this would be MM_{x-1} to MM_x), as intended by the step-wise migration. Again, changes of a previous evolution step δ_i with $i < min(S)$ are considered carried through, and changes any future evolution step δ_j with $j > max(S)$ are not yet considered at all. For example when creating $\mu'_{3,4,5}$ in Figure 7, the changes δ_1 and δ_2 are carried through, and changes δ_6 and δ_7 are disregarded for now. Only for changes δ_3, δ_4 and δ_5, a migration transformation can be created. This ensures the modularity of the migration transformations.

Technically, this degree of control is achieved by merging the metamodels $MM_{min(S)-1}$ and $MM_{max(S)}$ into a merged metamodel $MM_{min(S)-1,max(S)}$. This way, an in-place transformation can be created. Since in this context it is possible that a μ' is created for more than one δ, the merged metamodel can include more than one δ. The merging algorithm described above can be used recursively. For example if $S = (3,4,5)$ then $MM_{2,3,4,5} = merge(merge(merge(MM_1,\delta_2),\delta_3),\delta_4)$ is the metamodel used in the $\mu'_{3,4,5}$ in-place transformation. $MM_{2,3,4,5}$ is shown in Figure 9. Notice the cardinality relaxation of *SplitLink*, *LeftLink* and *RightLink*;

- *checkout transformation*: as a corrective mechanism, the full conformance is ensured of the partly migrated instance model to the partly evolved metamodel in the checkout transformation γ. This step is automatically achieved in our approach by applying the default migration transformations of the difference operations immediately after the customized migration step, i.e., $\gamma_i \circ \mu'_i$. After all, the default migration transformation is constructed so that its output models are syntactically correct. This way, e.g., instances of deleted metaclasses that are by accident not deleted by the customized migration transformation, are deleted by the checkout transformation, thereby ensuring conformance to the partly evolved metamodel. Typically however, the evolver has well-designed his/her customized migration transformation so that conformance is already ensured. The checkout transformation is merely a guarantee for conformance in the general case.

As M is composed of usual transformation models, it can be simply stored as any other transformation model. This way, future instance models conform to the old version can be migrated at a later instant.

3.2.3 Execution

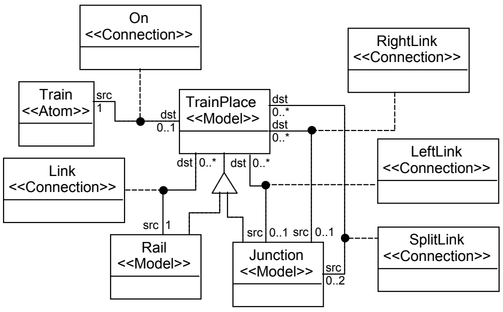


Figure 9: The merged metamodel $MM_{2,3,4,5}$ used for $\mu'_{3,4,5}$.

At first glance, the execution of the migration suite M is straightforward. On all instance models m, M is applied. More specifically a sequence of in-place transformations, like μ_i, μ'_j and γ_j, are applied in the given order. The ad hoc execution is not optimal however: in order for each of the in-place transformations to be executed, the instance model must be converted to that particular merged metamodel of the step. After execution, the result must be converted to the partly evolved metamodel. For example, a model m_5 conform to MM_5, must be converted first to $MM_{5,6}$. Then, the in-place transformation μ_6 can be applied, and the resulting model must be converted to metamodel MM_6. These conversions are trivial: a simple search/replace script on the data file of the instance model or a trivial transformation that implements a one-to-one mapping of elements can be automatically generated. However, this can cripple the execution performance of the migration. In Figure 10, a conversion is needed every time a different metamodel is used (i.e., a grey vertical line is crossed) throughout the execution of M. The top of Figure 10 represents the naive execution, requiring many conversions.

As a solution, after creation of migration transformation M, the different metamodels used in the in-place transformations are relaxed to the merged metamodel that spans all δ_i in Δ. All possible instance models throughout all migration steps can be expressed in the resulting metamodel MM_Δ. Every in-place transformation's used metamodel is changed to MM_Δ. Of course, this has to be done only once for M instead of for all instance models. With this optimized approach, an m that needs to be migrated only has to be converted twice: before applying the in-place transformations of M, and after applying the in-place transformations of M. In between, all artefacts use the same metamodel MM_Δ, and only in-place transformations are applied. The bottom of Figure 10 represents the optimized execution. The absence of model-to-model transformations adds to the execution performance of the migration because after evolution, it is probable that models only change slightly, if at all. If the evolver is confident in his/her customized migration transformation, he/she has the option to disable the execution of the checkout transformations, further improving the execution time of M.

4 Related Work

Co-evolution has been subject for research since the introduction of object-oriented database systems [BKKK87], consequently a huge amount of approaches dealing with this issue have been proposed (cf. [Rod92] for a survey). In this section we focus on most closely related approaches dedicated to reflecting changes of metamodels on models.

Sprinkle et al. [SK04] considered co-evolution of models by using changed semantics to de-

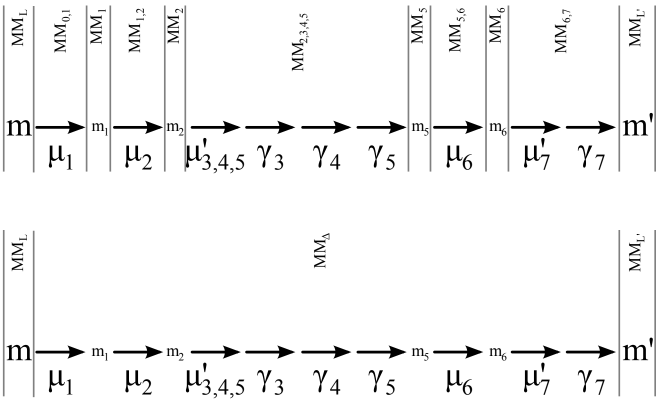


Figure 10: The naive execution needing a lot of conversions (top) and the optimized execution needing only two conversions (bottom).

sign co-evolution transformations. This differs from a syntactically driven approach that uses the metamodel deltas. In that work as well as in [SGM09], the authors proposed that syntactical co-evolution (where the importance is only to load, but not interpret, the models) is feasible automatically, but it seems to be impractical for semantic evolution. In the general case of semantic evolution concerns, semantics-preserving transformations must be developed by language engineers manually, based on their understanding of the semantic intent of the original models. However, for specific cases, semantically-preserving co-evolution transformations are possible.

Garces et al. [GJCB09] proposed a set of heuristics to automatically compute differences between two metamodel versions in order to adapt models. The computed differences are stored in a so-called matching model, acting as input for a higher-order transformation (HOT), producing a migration transformation. Cicchetti et al. [CDEP08] presented a similar approach, i.e., the approach is again based on a metamodel differences acting as input for a HOT.

In [Wac07], Wachsmuth proposed to combine ideas from object-oriented refactoring and grammar adaptation to provide the basis for automatic (meta)model evolution. In this respect, metamodel relations are defined based on M2M transformations, building the basis for the definition of semantics preservation and instance preservation.

Gruschko et al. [GKP07] tackled co-evolution of models by using M2M transformations by following a conservative copying algorithm. Conservative copying means that for initial model elements for which no transformation rule is found a default copy transformation rule is applied. This algorithm is implemented in model migration framework Flock [RKPP10].

In [NLBK09] the Model Change Language (MCL) is introduced. MCL is declarative and graphical language supporting a set of co-evolution idioms and conservative copying. Co-evolution rules going beyond the supported idioms have to be defined in terms of C++ code.

Finally, Herrmannsdoerfer et al. proposed COPE [HBJ09] for specifying the coupled evolution of metamodels and models. The co-evolution of metamodels and corresponding models is realized by a set of so-called coupled transactions, composing a whole co-evolution problem of modular transformations.

Summarizing, the differences between our proposed approach and the above mentioned approaches are twofold. First, we tackle co-evolution of models by employing existing in-place transformation languages instead of proposing dedicated co-evolution languages or reusing M2M transformation languages. Second, we refrain from developing the migration transformation for the whole metamodel evolution process at once. This incremental process is supported by computing intermediate merged metamodels, thus we allow to model the migration of models by ensuring all metamodel constraints. COPE also provides an incremental process, but in contrast to our approach, COPE uses a metamodel-independent representation of models. Thus, nothing can be said about the conformance of models during migration.

5 Conclusion

This paper presented a technique to deal with metamodel evolution and model co-evolution; despite the problem is an active field of research and a number of solutions have been proposed, several difficulties still demand for being alleviated. In particular, it has been illustrated a mechanism based on in-place migrations to reduce the accidental complexity of transformation design by shifting the focus on single co-evolutionary scenarios, in a step-by-step fashion. The evolver acts in a controlled environment which is narrowed down by the metamodel merging operation, which constraints her/his operative power and ensures syntactic consistency. Moreover, thanks to the in-place co-evolution unaffected instances are left untouched allowing, for example, the propagation of external links that would be lost after a re-creation of the same model element.

The approach enjoys a high degree of modularity, as relying on small co-evolution steps, which also results in an enhancement of re-use chances of the developed migration transformations. In fact, the technique permits to store both the manipulation a metamodel has been subject to and the corresponding countermeasures to re-establish the well-formedness of existing models.

Future investigations will be devoted to the analysis of the metamodel evolution representation and default migration transformations in order to further improve the degree of automation and re-use. In fact, on one hand the evolution representation should abstract from the context it has been observed; on the other hand, more that one co-evolutionary step could be adopted for the same metamodel evolution problem. Moreover, the approach will be extended to support the co-evolution of not only instance models, but also transformation models.

Bibliography

[BKKK87] J. Banerjee, W. Kim, H.-J. Kim, H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Record* 16(3):311–322, 1987.

[CDEP08] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio. Automating co-evolution in model-driven engineering. In *12th Int. EDOC Conf.* Pp. 222–231. 2008.

[CDP07] A. Cicchetti, D. Di Ruscio, A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology* 6(9):165–185, 2007.

[GJCB09] K. Garcés, F. Jouault, P. Cointe, J. Bézivin. Managing Model Adaptation by Precise Detection of Metamodel Changes. In *5th European Conf. on Model Driven Architecture - Foundations and Applications*. Pp. 34–49. Springer, 2009.

[GKP07] B. Gruschko, D. Kolovos, R. Paige. Towards synchronizing models with evolving metamodels. In *Int. Workshop on Model-Driven Software Evolution*. 2007.

[HBJ09] M. Herrmannsdoerfer, S. Benz, E. Juergens. COPE - Automating Coupled Evolution of Metamodels and Models. In *23rd ECOOP Conf.* Pp. 52–76. Springer, 2009.

[KMS$^+$09] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, M. Wimmer. Explicit Transformation Modeling. In *Models in Software Engineering*. Pp. 240–255. Springer, 2009.

[NLBK09] A. Narayanan, T. Levendovszky, D. Balasubramanian, G. Karsai. Automatic Domain Model Migration to Manage Metamodel Evolution. In *Model Driven Engineering Languages and Systems*. Pp. 706–711. Springer, 2009.

[Obj10] Object Management Group. Object Constraint Language Version 2.2. 2010.

[RG99] M. Richters, M. Gogolla. A metamodel for OCL. In *2nd Int. Conf. on the Unified Modeling Language*. Pp. 156–171. Springer, 1999.

[RKPP10] L. M. Rose, D. S. Kolovos, R. F. Paige, F. A. C. Polack. Model Migration with Epsilon Flock. In *3rd Int. Conf. on Theory and Practice of Model Transformations*. Pp. 184–198. Springer, 2010.

[Rod92] J. F. Roddick. Schema Evolution in Database Systems - An Annotated Bibliography. *SIGMOD Record* 21(4):35–40, 1992.

[SGM09] J. Sprinkle, J. Gray, M. Mernik. Fundamental Limitations in Domain-Specific Language Evolution. Technical report TR-090831, University of Arizona, 2009.

[SK04] J. Sprinkle, G. Karsai. A Domain-Specific Visual Language for Domain Model Evolution. *Journal of Visual Languages and Computing* 15(3-4):291–307, 2004.

[TJF$^+$09] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, J. Bézivin. On the Use of Higher-Order Model Transformations. In *5th European Conf. on Model Driven Architecture - Foundations and Applications*. Pp. 18–33. Springer, 2009.

[Wac07] G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *21rd European Conf. on Object-Oriented Programming*. Pp. 600–624. Springer, 2007.